

6

Programming Ideas

Adding Numbers

This section is about how to think and talk about the process of making a program. I developed the general approach while introducing elementary school children to computation. But the ideas that are good for children are good for other beginners, and perhaps for some experienced programmers. Variants of the example used here have been used with seventh graders, with college undergraduates, and with teachers. They illustrate a style of programming project, a style of programming language, and a meta-language or style of talking about programming as well as doing it. There is no suggestion that this style is uniquely correct. My message is on a different plane; I mean to assert the importance of paying more attention in the pedagogy of computation to such questions of style.

The problem is very recursive. I want to talk about programming, but I need to invent a way to talk about talking about programming! One way would be to give extracts from real dialog. But this is too cumbersome. Instead I shall condense real dialog into a kind of monolog about developing a program. The monolog gives an impression of one way I know how to think about developing a program. There is nothing very original about this way of thinking. The point I am making is about the technique of getting it out of our heads and into the pedagogy of teaching beginners.

In the discussion I carry with me a computational model in which there are little people, agents, experts in the computer that I can call on to help in thinking about the flow of my program, and, thus, in debugging my program. Keeping this model in mind helps me articulate what jobs need to be done and what procedures I need to get those jobs done. It also helps me figure out how these procedures interact with one another, how they report back what they have found out or constructed. Furthermore, as I debug my program and its individual procedures I talk again to these little people and get them to act out each procedure step by step, instruction by instruction.

The Project

We pretend the computer is ignorant of arithmetic and create an operation that will add two integers. No Logo arithmetic operations may be used. An apparent exception might seem to be `EQUALP (=)`, but it is used to compare

By Cynthia Solomon.

whether two Logo words or letters are the same. (It is an identity operator.) So +, <, >, COUNT, *, /, and REMAINDER are prohibited. Two implications arise from posing this project. One is that an addition operation can be decomposed into smaller procedures. The other is that numbers are really just words asked to play special roles.

This project generates interesting discussions. It really frees one's thinking about numbers and operations and primitiveness. It is true that arithmetic is a very necessary part of any computer's hardware, but the hardware is made up of "logical units" that are based on the same ideas we will investigate. How do computers really add? It's in their hardware. It's built into the system. It's hardwired. Is addition "hardwired" into *our* system? Are we like computers and so if a wire is loose we can't do it? What about addition among children? Is it really a built-in capacity, or are there pieces of knowledge that are acquired? Maybe we are so familiar with addition that we forget its components. In fact, addition must rely on lots of procedures.

Let's look at this project. Try to situate this particular task into a familiar environment. We have to imagine that there are no arithmetic operators available to us and that there are no arithmetic experts already existing in Logo. We want to make up an addition operation so that we can say

PRINT ADD 16 532

and the computer will say

548

Yes, addition is a familiar operation and it's easy for us to hand-simulate its job. But what if we had to tell a little person in the computer how to add? Where do we start? We might ask ourselves if we know of a similar experience. What we have to do is "teach the computer" to add—just as we might teach a person! Well, now, teachers teach kids to add; we were once those kids. How did we learn—can we give ourselves some tips? (But I thought it was hardwired and teacher just . . .)

At this point in past discussions two suggestions emerge. Teachers say we have to teach the computer the "number facts" and computerists say we have to build a 10×10 table. Great, I say, a beginning. I ask teachers how we teach the number facts and what are they and how many of them there are. I ask computerists if a 10×10 table is large enough and how we organize it. The teachers will face these issues too. After all, making a table is a way of "teaching" number facts.

What kind of table and what are number facts? A table of the sums of the first 100 numbers is very limited, and building a larger table is still very limited. Is that what I have in my head? Isn't there a key idea or two that I could build on without exhausting the computer's memory?

Do children learn "number facts" like $16 + 20 = 36$ as a primitive notion, or is there a more fundamental idea underlying it all? What do kids learn about numbers? They learn their relationship to each other. They learn to order them. *Sesame Street* teaches kids to count from 1 to 20. Kids learn to recognize the digits and their order. They learn that one is the name of 1 and eleven is the name of 11 and one hundred eleven is the name of 111. They learn that 11 is different from 2; they learn that 10 has been added to 1. But there is another way of discussing that change. Let's say 1

is a special word. We can create a new word by putting it together with another. So WORD 1 1 is 11, or eleven. Concatenating is a way of changing numbers.

Let's return to learning to recognize digits and ordering them. That indeed is what we have to tell the computer and build upon. You might say we want to teach the computer to count. On the other hand, it does us no good to see the computer spew out numbers from 1 to 500. We want the computer to know *how* to count. Think of what's involved in counting. How many symbols are there? In one sense there are ten, 0 1 2 3 4 5 6 7 8 9; but there are many constructions like 13 or 444; then there are also funny changes such as from 9 to 10, from 19 to 20, from 29 to 30, and so forth.

We want to teach the computer that 7 comes after 6 and 10 follows 9 and so on. Some of it is tricky. But look, the only elements used in a base-10 number system are 0 1 2 3 4 5 6 7 8 9. If kids learn how to use those ten symbols in thousands of different ways, surely we can teach the computer. There must be some rules that specify what to do to produce the "next number in sequence."

That's what we have to do. That is our plan of attack. Tell the computer what the basic elements (our data base) are. Then develop rules of behavior so that we can make the computer give us our number plus 1, that is, the next number. If the computer can do that, it knows how to count.

What is knowing how to count? Here's a computerist model: There is "in the head" a collection of little people, experts capable of doing a whole bunch of things like spewing numbers out, but also capable of conceiving questions like what comes after this or before that. The computer, like children, learns to recognize the digits, how to order them, and then how to use them to make other numbers.

Okay, let's make a procedure that knows about digits. For example, if it receives the input 3, it will output 4. It will add 1 (in some mysterious way) to its input.

```
ADD1 3 ---> 4
ADD1 7 ---> 8
```

We Make ADD1

There are a couple of ways (at least) to do this. People who suggested "teaching number facts" or making tables, of course, had the right idea. There are different ways of constructing tables. For example:

```
TO DIGITTABLE :DIGIT
IF :DIGIT = 0 [OP 1]
IF :DIGIT = 1 [OP 2]
IF :DIGIT = 2 [OP 3]
IF :DIGIT = 3 [OP 4]
IF :DIGIT = 4 [OP 5]
IF :DIGIT = 5 [OP 6]
IF :DIGIT = 6 [OP 7]
IF :DIGIT = 7 [OP 8]
IF :DIGIT = 8 [OP 9]
IF :DIGIT = 9 [OP 10]
END
```


We can also look at the ordered list of digits [0 1 2 3 4 5 6 7 8 9] as another representation that has the same effect if we have a NEXT type of operation.

```
NEXT 0 [0 1 2 3 4 5 6 7 8 9] ---> 1
```

NEXT will output the next element in the list after the one specified.

```
TO ADD1 :DIGIT
OP NEXT :DIGIT [0 1 2 3 4 5 6 7 8 9]
END
```

Why do I suggest this way? It is a more general method. This process will work for any base; all that needs to be changed are the elements of the list!

We Design NEXT

NEXT must supply ADD1 with a word. ADD1 will then send the word out as its answer. From the example of NEXT at work, we see that NEXT is given two inputs, a word like 0 and a list of words. NEXT tells its helpers to look for the word in the list. They send back the word following it in the list.

```
IF :WD = FIRST :LIST [OP FIRST BF :LIST]
```

If :WD doesn't match with :LIST's first word, one of NEXT's helpers just crosses the first word off the list and turns the job over to someone else.

```
OP NEXT :WD BUTFIRST :LIST
```

Check this procedure out.

```
TO NEXT :WD :LIST
IF :WD = FIRST :LIST [OP FIRST BF :LIST]
OP NEXT :WD BF :LIST
END
```

Notice there is a potential bug. What if :WD is not in :LIST? Let's remember the bug, but postpone dealing with it for the moment.

Let's try ADD1 now. Give it a thorough testing. You could exhaustively try each digit because there are only ten. Another strategy is to choose extremes like 0 and 9.

```
PR ADD1 0
1
PR ADD1 1
2
PR ADD1 2
3
PR ADD1 3
```

```
PR ADD1 9
FIRST DOESN'T LIKE [] AS INPUT IN NEXT
```


PROGRAMMING IDEAS

It's logical that there is a bug. After all, 9 is the last element of the list. So there is more work to be done; we have to teach ADD1 that $9 + 1 = 10$.

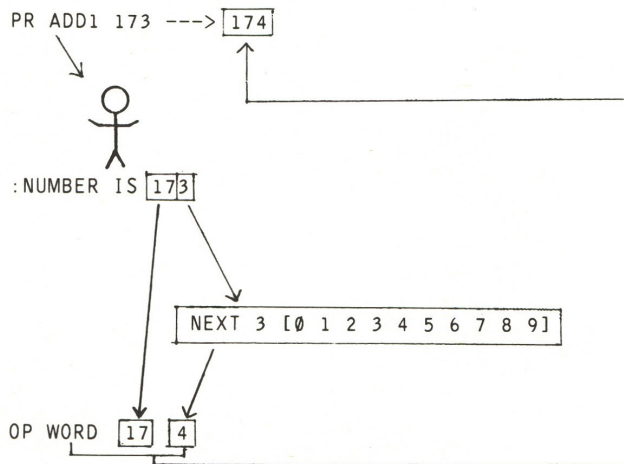
ADD1 will work on any number that doesn't end in 9 if we make one small change! Look, all numbers not ending in 9 behave like digits when you add 1 to them.

123 ---> 124
13 ---> 14

The only digit that changes is the LAST, so ADD1 merely makes up a new word by replacing LAST :DIGIT with NEXT LAST :DIGIT. Let's be opportunistic—seize the chance, change ADD1 and call its input NUMBER.

```
TO ADD1 :NUMBER
OP WORD [BL :NUMBER]
      NEXT [LAST :NUMBER] [0 1 2 3 4 5 6 7 8 9]
END
```

Now we trace through this procedure using the little person metaphor. As a reminder, I draw a stick figure.



(So we thought ADD1 was only good for nine inputs. Suddenly we see it's good for how many—millions? infinitely many? nine-tenths of all the numbers?)

Now ADD1 works on all numbers that don't end in 9. Would it work if we pretend 10 is a digit and add it to the list given NEXT—that is, [0 1 2 3 4 5 6 7 8 9 10]? Then

```
PR ADD1 9
10
```


but

```
PR ADD1 19
110
```

instead of 20!

So putting 10 in the list did not really help. This *nines* bug is not cured so quickly. This issue is really about what to do with the “carry” when adding numbers. If a number is 9 then the answer is 10, but if a number ends in 9 we want to carry one to add it to the next digit of the number. Now, how can wishful thinking help? How can we make use of what we just did? Let’s see how we do it. Try 179:

```
179 + 1 ---> 180
```

We turn the 9 into a 0 and add the 1 to the 17. We get 18... and don’t forget to glue the 18 and the 0 back together.

Make a special check for LAST :NUMBER being 9. Then replace the 9 by 0 and ADD1 to BL :NUMBER.

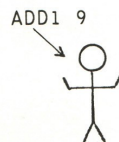
```
IF 9 = LAST :NUMBER [OP WORD ADD1 BL :NUMBER 0]
```

So

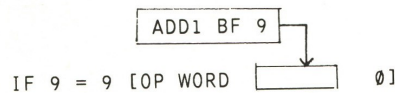
```
TO ADD1 :NUMBER
IF 9 = LAST :NUMBER [OP WORD ADD1 BL :NUMBER 0]
OP WORD BL :NUMBER NEXT LAST :NUMBER [0 1 2 3 4 5 6 7 8 9]
END
```

Now we try ADD1 with “little people.”

```
PRINT ADD1 9
```



```
:NUMBER IS 9
```



```
IF 9 = 9 [OP WORD
```

```
but BF 9 ---> "
```


PROGRAMMING IDEAS

We can fix this bug by making another special test

```
IF 9 = :NUMBER [OP 10]
```

as the first instruction in ADD1. Now

```
PR ADD1 179
180
```

and

```
PR ADD1 9999
10000
```

What luck! Perhaps you thought that the first 9 on the left would give trouble. But we lucked out (or were super smart!).

Adding Two Numbers

Now that we can add 1 to any number, we can really add any number to any other.

It's simple if we think of the kinds of procedures we know about. Some procedures operate on their inputs until they are empty or until a thing has been found. Other procedures do a job for a specified number of times. We can think of the next stage in our project as *adding one* to an input for a declared number of times.

6 + 4 is ADD1 ADD1 ADD1 ADD1 6.

Typically, counter procedures count down to 0 and then they know the job is done. But they use subtraction, and we are trying to invent addition without using any of Logo's built-in arithmetic operations. We can teach the computer to *subtract one*.

If we had a SUB1 procedure, then

```
TO ADDUP :NUM1 :NUM2
IF :NUM2 = 0 [OP :NUM1]
OP ADD1 ADDUP :NUM1 SUB1 :NUM2
END
```

Making a procedure for subtracting 1 is really easy because we have already thrashed through the difficulties encountered in ADD1. How can we use what we know about ADD1 to describe a SUB1? Let's look at a concrete situation.

```
PR SUB1 2
1
PR SUB1 9
8
PR SUB1 1
0
```

Can SUB1 use NEXT?

If we want NEXT 1 [...] to be 0, how should the list be ordered?
If we leave the list as [0 1 2 . . . 9], then NEXT 1 would output 2. It
should output 0. Reverse the list. Then NEXT 1 [9 8 7 6 5 4 3 2 1
0] outputs 0.

So

```
TO SUB1 :NUMBER
OP WORD BL :NUMBER NEXT LAST :NUMBER [9 8 7 6 5 4 3 2 1 0]
END
```

Try SUB1.

It works! As long as the numbers don't end in what? Nine is okay. Why?
The digit that is the LAST position of the list given to NEXT is the problem
digit. That is when a "carry" or a "borrow" takes place. So SUB1 must take
special measures when LAST :NUMBER is 0.

```
TO SUB1 :NUMBER
IF 0 = LAST :NUMBER [OP WORD SUB1 BL :NUMBER 9]
OP WORD BL :NUMBER NEXT LAST :NUMBER [9 8 7 6 5 4 3 2 1 0]
END
```

Now ADDUP works but very slowly and sometimes it needs too many people
to complete the job. Look, ADDUP 9999 9999 requires 9999 little peo-
ple.

Is there a shortcut? Yes. Let's treat the numbers as words and add the
LAST digit of each number to ADDUP until :N1 and :N2 have been added
together.

```
TO ADD :N1 :N2
IF EMPTY BL :N1 [OP ADDUP :N2 :N1]
IF EMPTY BL :N2 [OP ADDUP :N1 :N2]
OP WORD ADD BL :N1 BL :N2 ADDUP LAST :N1 LAST :N2
END
```

This is ideal but won't work very often. Do you know when it works?

```
PR ADD 34 21
55
PR ADD 2468 321
2789
```

but

```
PR ADD 19 19
218
```

The *carry* bug has to be dealt with. How can ADD tell if there is a carry? A
carry means that ADDUP will send back two digits (1 and something). That
makes it easy. ADD needs to test whether the result from ADDUP is one or
two digits long. ADD uses ADDIT to help and now looks like:

PROGRAMMING IDEAS

```

TO ADD :N1 :N2
  IF EMPTY? BL :N1 [OP ADDUP :N2 :N1]
  IF EMPTY? BL :N2 [OP ADDUP :N1 :N2]
  OP ADDIT ADDUP LAST :N1 LAST :N2 BL :N1 BL :N2
END

```

```

TO ADDIT :SUM :N1 :N2
  IF EMPTY? BF :SUM [OP WORD ADD :N1 :N2 :SUM]
  OP WORD ADD :N1 ADD1 :N2 BF :SUM
END

```

In some sense this project is completed. We have constructed an addition operation, and it works on positive integers. There are many extensions we could pursue. For example, handling negative numbers would probably necessitate making a subtract operation.

EXTENSIONS

In discussing setting up the table at the start, I mentioned the possibility of generalizing this scheme so that the operation would add numbers of other bases. What about fractions or decimals? But what about looking at a more general question? There are many arithmetic operations like MULTIPLY, DIVIDE, EXPONENTIATION, REMAINDER, BASE, CONVERSION, FACTORIAL. There are also others, like the Logo operation COUNT that outputs the length of a word or a list, and the predicates > (greater) and < (less). Any of these could be implemented as extensions to this project.

Although we might be able to write procedures to perform many of these operations, the process would probably be uncomfortably slow. This leads to the question: Are there some arithmetic operations that we couldn't define without special hardware or without special software? What operations are primitive? Imagine writing WORD or LIST or FIRST or BUTFIRST. What would be required? Is the derivation too clumsy? The answers to these questions will undoubtedly change as the contexts in which they arise change.

PROGRAM LISTING

<pre> TO ADD :N1 :N2 IF EMPTY? BL :N1 [OP ADDUP :N2 :N1] IF EMPTY? BL :N2 [OP ADDUP :N1 :N2] OP ADDIT ADDUP LAST :N1 LAST :N2 BL :N1 BL :N2 END </pre>	<pre> TO ADDUP :NUM1 :NUM2 IF :NUM2 = 0 [OP :NUM1] OP ADD1 ADDUP :NUM1 SUB1 :NUM2 END </pre>
<pre> TO ADDIT :SUM :N1 :N2 IF EMPTY? BF :SUM [OP WORD ADD :N1 :N2 :SUM] OP WORD ADD :N1 ADD1 :N2 BF :SUM END </pre>	<pre> TO ADD1 :NUMBER IF 9 = :NUMBER [OP 10] IF 9 = LAST :NUMBER [OP WORD ADD1 BL :NUMBER 0] OP WORD BL :NUMBER NEXT LAST :NUMBER [0 1 2 3 4 5 6 7 8 9] END </pre>

```

TO NEXT :WD :LIST
IF :WD = FIRST :LIST [OP FIRST BF ►
    :LIST]
OP NEXT :WD BF :LIST
END

```

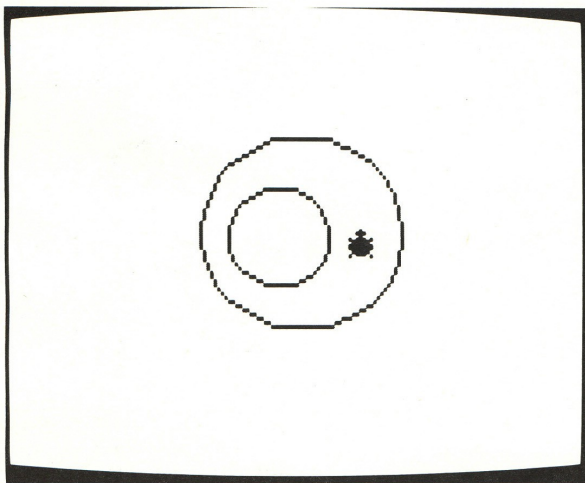
```

TO SUB1 :NUMBER
IF 0 = LAST :NUMBER [OP WORD SUB1 BL ►
    :NUMBER 9]
OP WORD BL :NUMBER NEXT LAST :NUMBER ►
    [9 8 7 6 5 4 3 2 1 0]
END

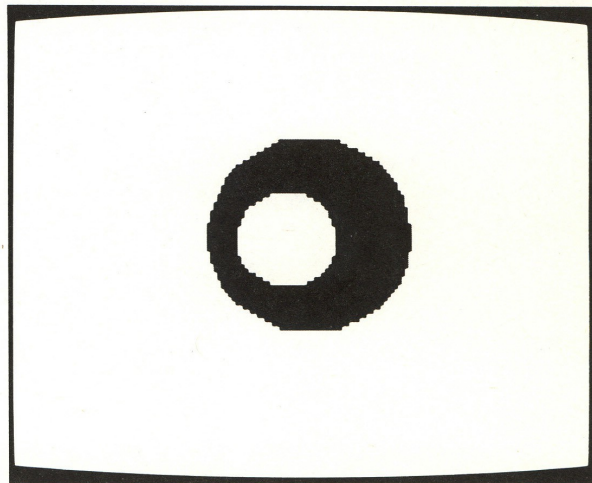
```

Fill

FILL is a program to fill in solid areas on the graphics screen.



Before



After

Figure 1

To use FILL, position the turtle inside the area you want to fill. Then type the command FILL with no inputs. The area the program will fill is bounded by lines drawn with any pen.* For example, try this:

```

CS
REPEAT 4 [FD 80 RT 90]
PU
SETPOS [20 20]
FILL

```

*If the screen dot at the turtle's position was already drawn with one of the pens, then FILL treats that pen as the background color for filling. So if you have a filled-in area on the screen, you can draw a picture within that area and fill the inside of the picture using another color.

By Brian Harvey.

PROGRAMMING IDEAS

to draw a solid, filled-in square. The SETPOS instruction is necessary to position the turtle inside the square, rather than on its edge, before using FILL.

Note: If you have a 16K Atari computer, you should use the number 8192 instead of 16384 in procedure POSADDR.

How It Works: Overview

Figure 2 shows a sort of eccentric doughnut with the turtle positioned between the two circles, so that the doughnut shape will be filled. The program begins by filling horizontally from the turtle's initial position, in both directions (figure 3). It remembers how far it got, to set left and right limits for what comes later. Then it starts moving up (figure 4), filling horizontally at each level.

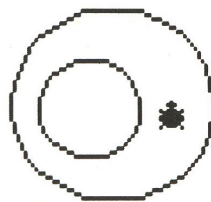


Figure 2

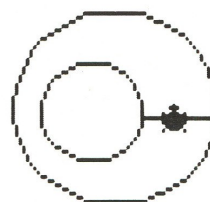


Figure 3

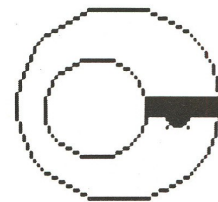


Figure 4

But when a newly filled line extends beyond the previous line (as illustrated by the left edge of the filled area in figure 4), the program also checks for an unfilled space below the new horizontal stretch. If it finds one, it starts filling downward in that new area (figure 5). This search for new areas works from left to right on each line, so (figure 6) the program continues moving downward below the inner hole until it reaches the bottom (figure 7).

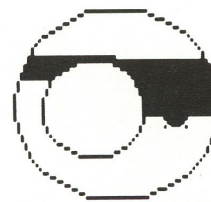


Figure 5

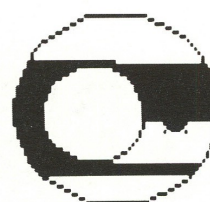


Figure 6

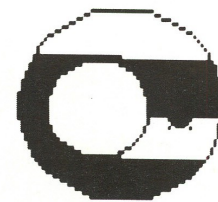


Figure 7

Then it starts moving up into the newly discovered area to the right of the hole (figure 8), and when that area is filled, the program continues its interrupted upward filling of the top area (figure 9). The final result is shown in figure 10.

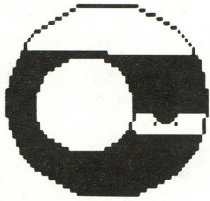


Figure 8

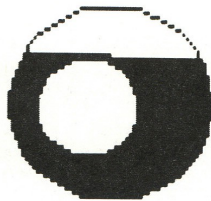


Figure 9

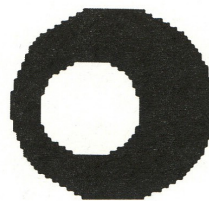


Figure 10

Screen Coordinates and Turtle Steps

The graphics screen consists of about 15,000 small dots, in a rectangular array of 96 rows and 160 columns. Logo draws lines on the screen by “turning on” some of these dots. To fill an area, we must also turn on dots in this array.*

When you use the FORWARD command, the distance measured in “turtle steps” is not the same as the number of screen dots (or *pixels*) through which the turtle passes. There are two reasons for this difference. The first reason is that the distance between two vertically adjacent pixels is greater than the distance between two horizontally adjacent pixels. If Logo measured distances in pixels, squares would come out looking like tall rectangles. Instead, Logo uses the *aspect ratio* (the ratio of a horizontal pixel distance to a vertical pixel distance) as a scale factor for vertical turtle steps. The second reason is that both vertical and horizontal turtle steps are scaled by a factor of two, so that 100 turtle steps is a reasonable distance on the screen.

The reason this scaling of distances is important for the FILL project is that we’re going to have to think in terms of pixels, not in terms of turtle steps. Remember that the overall task of the program is to move along the screen looking for the border of the region we want to fill. In other words, the program must look at a position on the screen to see if that position is in the background color. If so, the program should fill in that position and move on to the next. Suppose we wrote the program in terms of turtle steps. (We’d then use FORWARD 1 to move from one position to the next.) Since a turtle step is smaller than the distance between pixels, two consecutive turtle positions will often occupy *the same pixel* on the screen! After filling in the first position, we’d move on to the next position and think we’d

*For more details about the screen array, see the Savepict and Loadpict project.

hit the border, because the screen dot would no longer be in the background color.

The approach I took in writing `FILL` is to think about positions in terms of screen pixel coordinates, rather than turtle coordinates. The top-level procedure `FILL` computes the pixel coordinates corresponding to the turtle's position, and those pixel coordinates are used as inputs to the lower-level procedures which do the real work. Figure 11 shows the screen coordinate system used in `FILL`. The origin of this system (the point with horizontal and vertical coordinates zero) is in the top left corner of the screen. `XCOR` (the horizontal coordinate) gets bigger as you move to the right. `YCOR` (the vertical coordinate) gets bigger as you move *down* the screen; compare this with Logo's turtle-step `YCOR`, which gets bigger as you move up the screen.

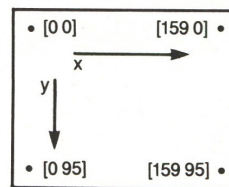


Figure 11

Because `FILL` uses screen coordinates instead of turtle coordinates, we can't use the usual Logo graphics procedures like `FORWARD` or `XCOR`. Instead, we have to write our own tools for examining and modifying screen pixels. Two important procedures in this project are `COLOR.AT`, which examines the color of a pixel, and `DOT`, which fills in a pixel.

One final point about the screen array is that each byte of computer memory contains the color information for four pixels. Logo's `EXAMINE` procedure lets us look at an entire byte at a time, not just one pixel. Therefore, the program is more efficient if we can design it to examine four pixels at once. You'll see how we do that when we get to the description of the `FILL.RAY` procedure.

Initialization

Procedures `FILL`, `FILL1`, and `FILL2` are invoked just once each time you use `FILL`. They set up certain information that is needed throughout the program. Here are the procedures, followed by a list of their important variables.

```
TO FILL
  IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH 0.8]
  PU
  FILL1 79+INT (XCOR/2) 48-(INT (YCOR*:SCRUNCH/2))
    IF (PEN="PE) [0] [PN + 1]
END

TO FILL1 :XCOR :YCOR :PEN
  FILL2 COLOR.AT :XCOR :YCOR 0 0
END
```

```

TO FILL2 :BG :BGBYTE :PENBYTE
MAKE "BGBYTE 85*:BG
MAKE "PENBYTE 85*:PEN
FILL.BOTH FILL.LINE 0 0
END

```

- SCRUNCH The aspect ratio. This ratio is 0.8 unless you have changed it by using Logo's `SETSCR` command. There is no direct way for `FILL` to find out the current aspect ratio, so it simply assumes a value of 0.8 unless you provide a different value in the global variable named `SCRUNCH` before you use `FILL`. This information is used in the procedure `FILL` to help convert the current turtle position into screen pixel coordinates.
- XCOR The turtle's current horizontal position, in pixels. Note that the *variable* `XCOR` is different from the Logo *procedure* named `XCOR`, which operates in turtle steps. Note also that the name `XCOR` is used for other variables in several sub-procedures to hold local position information.
- YCOR The turtle's current vertical position, in pixels. The same notes apply as for `XCOR`.
- PEN The pen we should use for filling. Since one of the possibilities is to fill by erasing (setting pixels to the background color), we don't use exactly the same numbers that Logo uses for pens. Instead, Logo's pens 0 to 2 are represented in this variable with the numbers 1 to 3, while the number 0 represents the background color. We use the background color if the turtle is in `penerase` (`PE`) when you give the `FILL` command. Representing the background as 0 and the three pens as 1 to 3 is convenient in this program, because those numbers are the ones that are actually stored in the screen memory in the Atari computer.
- BG The pen number that is the background of the region we should fill. This is not necessarily *the* background color of the screen. When you give the `FILL` command, `FILL1` uses sub-procedure `COLOR.AT` to find out whether the particular pixel at the turtle's position is in the background color or in one of the three pens. Whichever is true of that pixel, the corresponding color is what we look for to determine the region we're supposed to fill. The value of `BG` is coded like that of `PEN`: 0 for background, 1 to 3 for the three pens.
- BGBYTE `FILL2` sets this variable to the value of `BG` multiplied by 85. This has the effect of reproducing the value of `BG` four times in a byte.* A memory byte that contains this number represents four consecutive `BG`-colored pixels.
- PENBYTE This is `PEN` reproduced four times in a byte, and it represents four consecutive `PEN`-colored pixels.

*If you understand how numbers are represented in binary in the computer's memory, you'll want to know that 85 is 01010101 binary. Multiplying a two-bit code (the possible values are 0 to 3) by this number has the desired effect of reproducing it four times in the eight-bit byte. If you don't know about binary representation, don't worry about it.

PROGRAMMING IDEAS

There is a trick in the way FILL1 calls FILL2. FILL2 has three inputs, named BG, BGBYTE, and PENBYTE. FILL1 provides the real value for the first input (BG), but it uses zero as the values for the others.

```
FILL2 (COLOR.AT :XCOR :YCOR) 0 0
```

FILL2 starts by assigning new values to these input variables. The reason for this trick is to make BGBYTE and PENBYTE *local* variables of FILL2 instead of global variables. Using local variables avoids leaving clutter around when FILL is finished. Actually, the use of local variables isn't terribly important in this particular example, but the same trick is used in some procedures we'll see later (most notably FILL.UP1) where it really is essential.

FILL2 begins the real work of filling an area with the instruction

```
FILL.BOTH FILL.LINE 0 0
```

FILL.LINE fills horizontally, on the line where the turtle is when you give the FILL command. Then FILL.BOTH uses information output by FILL.LINE to handle the vertical part of the filling. We'll discuss these procedures in more detail in the following sections.

Filling a Line

Here is the definition of FILL.LINE.

```
TO FILL.LINE :LEFT :RIGHT
MAKE "LEFT FILL.RAY :XCOR :YCOR (-1)
MAKE "RIGHT FILL.RAY :XCOR+1 :YCOR 1
OP (SE :YCOR :LEFT :RIGHT)
END
```

This procedure uses the same trick as FILL2 to create local variables LEFT and RIGHT. Although they're defined as inputs to FILL.LINE, these variables really get their values within FILL.LINE itself.

Most Logo procedures are either *commands*, which do something visible like move a turtle, or *operations*, which have no visible effect but instead output a value, like the arithmetic operations. FILL.LINE has both an effect and an output. Its effect is to fill the line on which the turtle starts. (Turn back to figure 3 to see FILL.LINE at work.) Its output is a list of coordinates, indicating how far to the left and right it was able to fill.

The turtle starts out somewhere in the middle of the area we want to fill. To fill the line containing the turtle's position, we have to start from that position and fill both to the left and to the right. FILL.LINE invokes FILL.RAY twice, first to fill toward the left and then to fill toward the right. FILL.RAY knows which direction to use because of its third input, which is -1 to fill leftward or 1 to fill rightward.

Filling in One Direction

FILL.RAY does all of the actual filling in of dots in the entire FILL program. The other procedures simply figure out where to tell FILL.RAY to go to work.

Because of the importance of FILL.RAY, I put a lot of effort into trying to make it fast. Unfortunately, the cost of speed is complexity. Let's start by examining a version of FILL.RAY that doesn't yet have all of the efficiency features added.

```
TO FILL.RAY :XCOR :YCOR :DELTA
IF EDGE :XCOR :YCOR [OP :XCOR-:DELTA]
DOT :XCOR :YCOR
OP FILL.RAY :XCOR+:DELTA :YCOR :DELTA
END
```

FILL.RAY has three inputs. The first two are the horizontal (x) and vertical (y) screen coordinates of the pixel at which we want to start filling. The third input tells FILL.RAY the direction in which to fill.*

The strategy of FILL.RAY is this:

1. Look at a pixel to see if it's in our background color.[†]
2. If it's not in our background color, it is a border for the area we're filling. Output the x coordinate of the last pixel we actually filled—the one before this one.
3. If it is in our background color, fill it and move on to the next pixel in the desired direction, left or right.

To implement this strategy, FILL.RAY uses two subprocedures. The first, EDGE, is a predicate that outputs TRUE if the pixel it examines is in something *other than* the background color. The second subprocedure, DOT, fills in the pixel at the coordinates you give it as inputs. We'll look at those procedures later. For now, the important point is to understand how they're used by FILL.RAY.

Filling Vertically

We have seen how the FILL program fills one horizontal line, the one containing the turtle's position. What remains is to fill more lines, above and below that first one. This task is entrusted to FILL.BOTH.

```
TO FILL.BOTH :RANGE
FILL.UP :RANGE (-1)
FILL.UP :RANGE 1
END
```

*The word *delta* is the name of a Greek letter (Δ) that is often used in mathematics to represent a *change* in something. In this case, :DELTA is added to :XCOR each time a dot is filled in. If :DELTA is positive, the new x coordinate is to the right of the old one. If :DELTA is negative, the new coordinate is to the left.

[†]As explained earlier, this may or may not be *the* background color of the screen.

PROGRAMMING IDEAS

The name `FILL.BOTH` indicates that it must fill both above and below the line we've already filled. Just as `FILL.LINE` invokes `FILL.RAY` twice, `FILL.BOTH` invokes a subprocedure called `FILL.UP` twice.

`FILL.BOTH`, you'll remember, is invoked by `FILL2`. The input to `FILL.BOTH` is the output from `FILL.LINE`. This output is a list of three numbers: the vertical (y) coordinate of the line we've filled, and the leftmost and rightmost horizontal (x) coordinates of the line.* See figure 12 for a pictorial representation of this information.

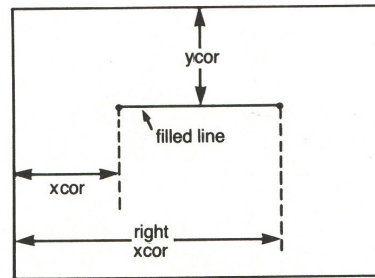


Figure 12

`FILL.BOTH` gives two inputs to `FILL.UP`. The first input is the range list. The second input tells `FILL.UP` the direction (up or down) in which to fill. This second input is either 1 or -1, just like the similar direction input to `FILL.RAY`.

Here is the definition of `FILL.UP`.

```
TO FILL.UP :RANGE :DELTA
FILL.UP1 (:DELTA+FIRST :RANGE)
  FIRST BF :RANGE LAST :RANGE :DELTA 0 0
END
```

All it does is to invoke `FILL.UP1`, with six inputs. The first three inputs are the three members of the range list, except that the vertical coordinate is offset by one. (The reason is this: the range list output by `FILL.LINE` contains the vertical coordinate of the line it just filled. We now want to fill a new line, just above or just below that line. The first input to `FILL.UP1` is the vertical coordinate of the line we should fill next.) The fourth input to `FILL.UP1` is the direction indicator, 1 or -1. The fifth and sixth inputs are given as zero. They're really used as local variables within `FILL.UP1`.

The Smart Procedure

`FILL.UP1` really contains all the geometric knowledge of this program. `FILL.UP1` has to know how to fill an area above or below a given line. This task would be very easy if areas were always pleasantly shaped. In fact, though, the filling job may have to "double back" because of irregularities in the area we're filling. This complication is illustrated in figures 4 and 5

*If you want to be picky, of course, what we've filled is a line *segment*, not a line.

(reproduced here). In figure 4, we are filling upward. This process continues straightforwardly until we get above the "hole" in the center of the region. At that point, the program is able to extend the filled area farther to the left. It then discovers a new, unfilled region below the new line. Figure 5 shows that the program has reversed its direction; it's filling downward to take care of the area to the left of the central hole.

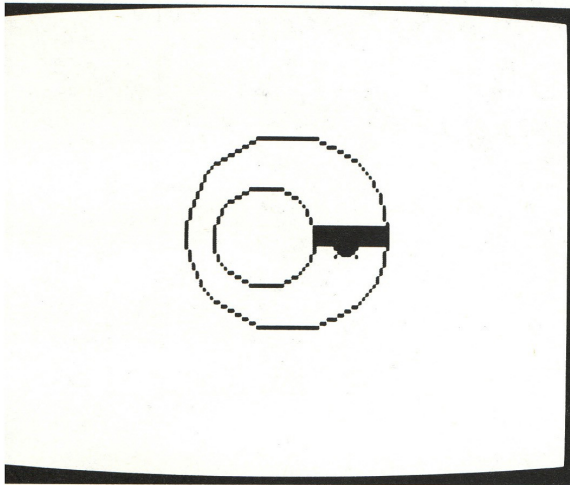


Figure 4

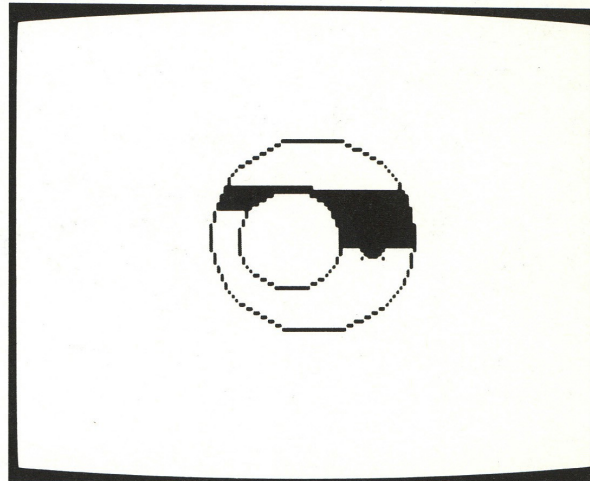


Figure 5

The strategy of FILL.UP1 is quite complicated, but it's made up of two kinds of parts: using FILL.RAY, and using FILL.UP1 recursively.

1. Use FILL.RAY to fill at the current vertical position.
2. Compare the horizontal extent of FILL.RAY's work to the horizontal extent of the previous line.
3. If we've gone farther on this line than on the previous line, invoke FILL.UP1 recursively to deal with the area newly exposed.
4. Also invoke FILL.UP1 recursively to continue with the same region we were already filling.

Since the procedure is complicated, we'll show its definition with the instruction lines numbered. In the discussion that follows we'll refer to particular lines by number.

```
[1] TO FILL.UP1 :YCOR :LEFT :RIGHT :DELTA :NEWL :NEWR
[2] MAKE "NEWL FILL.RAY :LEFT :YCOR (-1)
[3] IF :NEWL<:LEFT
    [FILL.UP1 :YCOR-:DELTA :NEWL :LEFT (-:DELTA) 1 0]
[4] MAKE "NEWR
    IF :NEWL>:RIGHT [ :NEWL-1] [FILL.RAY :LEFT+1 :YCOR 1]
[5] IF :NEWL<:NEWL+1
    [FILL.UP1 :YCOR+:DELTA :NEWL :NEWR :DELTA 2 0]
[6] IF :NEWR>:RIGHT
    [FILL.UP1 :YCOR-:DELTA :RIGHT :NEWR (-:DELTA) 3 0]
[7] MAKE "NEWL FIND.BG :NEWR :YCOR :RIGHT
[8] IF WORDP :NEWL [FILL.UP1 :YCOR :NEWL :RIGHT :DELTA 4 0]
[9] END
```


PROGRAMMING IDEAS

Refer to figure 13 for a picture of what happens in `FILL.UP1`'s work. The solid horizontal line in that picture was filled earlier, either by `FILL.LINE` or by the previous invocation of `FILL.UP1`. The dashed horizontal line above is the one that will be filled by the current invocation of `FILL.UP1`.

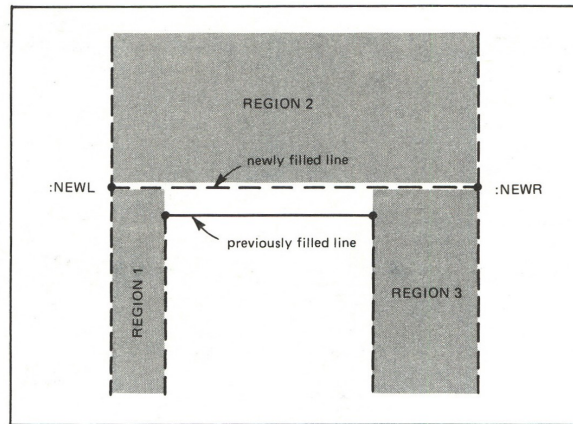


Figure 13

Here is a list of the variables used in `FILL.UP1`.

YCOR	The vertical coordinate of the dashed line, the one being filled by this invocation of <code>FILL.UP1</code> .
LEFT	The leftmost horizontal coordinate of the <i>solid</i> line, the one previously filled.
RIGHT	The rightmost horizontal coordinate of the solid, previously filled line.
DELTA	The direction indicator. Its value will be 1 if the new (dashed) line is above the old (solid) line, or -1 if the new line is below the old line.
NEWL	The leftmost horizontal coordinate of the new (dashed) line.
NEWR	The rightmost horizontal coordinate of the new line.

Each invocation of `FILL.UP1` actually fills only one line. This filling is done by using `FILL.RAY` twice, on lines 2 and 4 of the procedure. Line 2 fills to the left of `:LEFT`, and line 4 fills to the right of `:LEFT`. The variables `NEWL` and `NEWR` are given as values the x coordinates of the endpoints of the newly filled line.

When we're filling vertically, the most obvious thing is that after filling one line, we must continue filling vertically in the same direction. Referring to figure 13, after filling the dashed line we must continue upward, filling region 2 in the figure. (Of course, we don't know yet what the exact shape of that region will be. In the figure, it's shown as extending straight up, but the edges might really be curved.) This continuation in the same vertical direction is done in line 5 of the procedure.

How do we know when to stop? The answer is that if on *this* level we didn't manage to fill anything (because we ran into borders right away), then we shouldn't continue to the next level up. That's why line 5 compares `:NEWL` to `:NEWR`. If they're equal, we didn't fill anything on this level.

There are two possible cases of “doubling back”: one if the newly filled line extends farther to the left than the old line, and one if the new line extends farther to the right. In figure 13, both of these situations have arisen.

We know that the new line has extended farther to the left than the old line if `:NEWL` is less than `:LEFT`. This is the situation at the transition from figure 4 to figure 5, which we’ve discussed earlier. Line 3 of the procedure checks for this situation. If the condition is met, then `FILL.UP1` is recursively invoked to fill what is labeled region 1 in figure 13.

Similarly, we must double back on the right (into region 3 of figure 13) if `:NEWR` is greater than `:RIGHT`. Line 6 of `FILL.UP1` takes care of this case. An example of this situation is at the transition between figure 7 and figure 8 (reproduced here). In figures 6 and 7, the program was filling downward. When the lower boundary of the region is reached, in figure 7, the program doubles back and starts filling upward in figure 8.

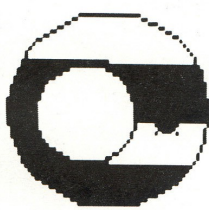


Figure 6

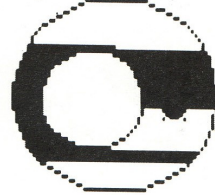


Figure 7

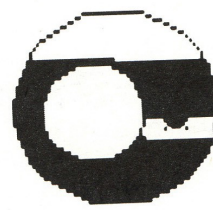


Figure 8

By the way, the doubling back into region 1 happens *before* the continued filling of region 2. But the doubling back into region 3 happens *after* region 2 is filled. That’s because lines 3, 5, and 6 happen to be in the order they are. If line 3 were moved below line 5, the program would always complete one direction of filling before starting in the other direction.

There is one more complication in `FILL.UP1`. The line that is filled in lines 2 and 4 of the procedure extends to both sides of `:LEFT`, the leftmost end of the previously filled line. Suppose that a border is reached above the old line, before its rightmost end. This situation is shown in figure 14. Since we want to fill all of the area above the previously filled line, it’s not enough to fill the area above the dashed line in the figure. We must also fill what is labeled as region 4.

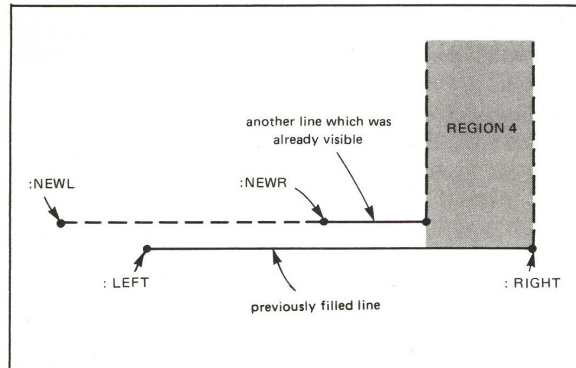


Figure 14

PROGRAMMING IDEAS

How do we know when this situation arises? First of all, :NEWB must be less than :RIGHT. Second, if we look to the right of :NEWB, we must find another patch of background color before reaching :RIGHT. This search is conducted by FIND.BG, which is used on line 7 of FILL.UP1. FIND.BG outputs the empty list if it does not find a suitable background pixel. If it does find one, FIND.BG outputs the x coordinate of that pixel. This coordinate is the left edge of region 4. Line 8 of FILL.UP1 checks to see if FIND.BG found a background pixel. If so, it invokes FILL.UP1 once more to fill region 4.

Examining a Screen Pixel

The real core of this program is the strategy FILL.UP1 uses to explore the nooks and crannies of irregular shapes. What remains for us to consider are the utility procedures that actually manipulate individual pixels. For example, FILL.RAY relies on EDGE to find out whether a particular pixel is a border of the area.

```
TO EDGE :XCOR :YCOR
OP NOT EQUALP :BG COLOR.AT :XCOR :YCOR
END
```

```
TO COLOR.AT :XCOR :YCOR
OP PIXEL (.EXAMINE POSADDR :XCOR :YCOR) REMAINDER :XCOR 4
END
```

```
TO POSADDR :XCOR :YCOR
OP 16384 + 40* :YCOR + INT (:XCOR/4)
    Use 8192 instead of 16384 for 16K Atari.
END
```

```
TO PIXEL :BYTE :XCOR
IF :XCOR=0 [OP INT (:BYTE/64)]
IF :XCOR=1 [OP REMAINDER INT (:BYTE/16) 4]
IF :XCOR=2 [OP REMAINDER INT (:BYTE/4) 4]
OP REMAINDER :BYTE 4
END
```

EDGE compares the color* of a particular pixel with our background color. It outputs TRUE if the two are different. That is, EDGE outputs TRUE if the pixel it's examining is on an edge of the area we're filling.

COLOR.AT outputs the color status of a pixel. Remember that each byte of screen memory contains this information for four pixels. So COLOR.AT must read a byte of screen memory and extract from that byte the particular pixel we're interested in.

POSADDR translates from the x and y coordinates of a pixel to the byte address in screen memory that contains that pixel. If you want to know about how these addresses are calculated, read the Savepict and Loadpict project.

*Actually, not the color number, but the pen number, in the form discussed earlier in the description of the PEN and BG variables.

PIXEL extracts one pixel from a byte. It takes two inputs. The first input is a byte of screen memory. The second input is a number from 0 to 3, specifying which pixel we want within that byte.

Filling One Pixel

FILL.RAY uses the procedure DOT to fill each pixel. DOT takes the coordinates of the pixel as inputs. Here it is.

```

TO DOT :XCOR :YCOR
DOTA POSADDR :XCOR :YCOR
END

TO DOTA :ADDR
RUN SE (WORD "DOT REMAINDER :XCOR 4) .EXAMINE :ADDR
END

TO DOT0 :BYTE
.DEPOSIT :ADDR SUM (REMAINDER :BYTE 64) 64*:PEN
END

TO DOT1 :BYTE
.DEPOSIT :ADDR (SUM (64*INT (:BYTE/64))
(16*:PEN) (REMAINDER :BYTE 16))
END

TO DOT2 :BYTE
.DEPOSIT :ADDR (SUM (16*INT (:BYTE/16))
(4*:PEN) (REMAINDER :BYTE 4))
END

TO DOT3 :BYTE
.DEPOSIT :ADDR SUM (4*INT (:BYTE/4)) :PEN
END

```

DOT must change the color of one pixel in a byte, leaving the other three pixels of that byte unchanged. Since Logo's .DEPOSIT command can only change an entire byte of memory at once, DOT has to combine the new color of one pixel with the old colors of the three other pixels. Precisely how to do this depends on which pixel in the byte we want to change, so DOT has a subprocedure for each possibility. These subprocedures are named DOT0 through DOT3.

Making FILL.RAY More Efficient

Earlier we looked at a simplified version of FILL.RAY, which examines and fills one pixel at a time. It's faster if we can examine an entire byte full of pixels at once. Here is the modified FILL.RAY, which does that, along with some new subprocedures.

PROGRAMMING IDEAS

```

TO FILL.RAY :XCOR :YCOR :DELTA
IF BYTEPOS :XCOR :DELTA
  [IF :BGBYTE=.EXAMINE POSADDR :XCOR :YCOR
    [OP FILL.CHUNK :XCOR :YCOR
      POSADDR :XCOR :YCOR :DELTA] ]
IF EDGE :XCOR :YCOR [OP :XCOR-:DELTA]
DOT :XCOR :YCOR
OP FILL.RAY :XCOR+:DELTA :YCOR :DELTA
END

TO FILL.CHUNK :XCOR :YCOR :ADDR :DELTA
.DEPOSIT :ADDR :PENBYTE
IF :BGBYTE=.EXAMINE :ADDR+:DELTA
  [OP FILL.CHUNK :XCOR+4*:DELTA :YCOR
    :ADDR+:DELTA :DELTA]
OP FILL.RAY :XCOR+4*:DELTA :YCOR :DELTA
END

TO BYTEPOS :XCOR :DELTA
IF :DELTA>0 [OP 0=REMAINDER :XCOR 4]
OP 3=REMAINDER :XCOR 4
END

```

FILL.RAY can only examine a complete byte of four pixels if the pixel it's ready to examine next is the first one in a byte. The predicate BYTEPOS outputs TRUE if that is the case. If not, FILL.RAY does the same things it did in the simpler version.

If BYTEPOS is TRUE, FILL.RAY examines the entire byte containing the pixel of interest. If that byte contains four pixels all in background color, we can fill all four at once. The variable BGBYTE contains the byte value that represents four background pixels.

If FILL.RAY does find a byte full of background pixels, it uses FILL.CHUNK to fill all four at once. FILL.CHUNK then examines the next byte to see if it, too, contains four background pixels. Once FILL.CHUNK reaches a byte that is not entirely background, it reverts to the use of FILL.RAY to check individual pixels.

Finding Region 4

The procedure FIND.BG, which is used to detect the appearance of a fourth region to fill, is very much like FILL.RAY, with two exceptions. First, FIND.BG passes over nonbackground pixels and stops when it reaches a background pixel. Second, FIND.BG just examines the pixels, whereas FILL.RAY fills them also.

```

TO FIND.BG :XCOR :YCOR :LIMIT
IF :XCOR>:LIMIT [OP []]
IF BYTEPOS :XCOR 1
  [IF :PENBYTE=.EXAMINE POSADDR :XCOR :YCOR
    [OP FIND.BG :XCOR+4 :YCOR :LIMIT] ]
IF NOT EDGE :XCOR :YCOR [OP :XCOR]
OP FIND.BG :XCOR+1 :YCOR :LIMIT
END

```

PROGRAM LISTING

```

TO FILL
IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH ►
    0.8]
PU
FILL1 79+INT (XCOR/2) 48-(INT ►
    (YCOR*SCRUNCH/2)) IF (PEN="PE) ►
    [0] [PN + 1]
END

TO FILL1 :XCOR :YCOR :PEN
FILL2 COLOR.AT :XCOR :YCOR 0 0
END

TO FILL2 :BG :BGBYTE :PENBYTE
MAKE "BGBYTE 85*BG
MAKE "PENBYTE 85*PEN
FILL.BOTH FILL.LINE 0 0
END

TO FILL.LINE :LEFT :RIGHT
MAKE "LEFT FILL.RAY :XCOR :YCOR (-1)
MAKE "RIGHT FILL.RAY :XCOR+1 :YCOR 1
OP (SE :YCOR :LEFT :RIGHT)
END

TO FILL.BOTH :RANGE
FILL.UP :RANGE (-1)
FILL.UP :RANGE 1
END

TO FILL.UP :RANGE :DELTA
FILL.UP1 (:DELTA+FIRST :RANGE) FIRST ►
    BF :RANGE LAST :RANGE :DELTA 0 0
END

TO FILL.UP1 :YCOR :LEFT :RIGHT :DELTA ►
    :NEWL :NEWL
MAKE "NEWL FILL.RAY :LEFT :YCOR (-1)
IF :NEWL<:LEFT [FILL.UP1 :YCOR-:DELTA ►
    :NEWL :LEFT (-:DELTA) 1 0]
MAKE "NEWL IF :NEWL>:RIGHT [:NEWL-1] ►
    [FILL.RAY :LEFT+1 :YCOR 1]
IF :NEWL<:NEWL+1 [FILL.UP1 ►
    :YCOR+:DELTA :NEWL :NEWL :DELTA 2 ►
    0]
IF :NEWL>:RIGHT [FILL.UP1 :YCOR-:DELTA ►
    :RIGHT :NEWL (-:DELTA) 3 0]
MAKE "NEWL FIND.BG :NEWL :YCOR :RIGHT
IF WORDP :NEWL [FILL.UP1 :YCOR :NEWL ►
    :RIGHT :DELTA 4 0]
END

TO EDGE :XCOR :YCOR
OP NOT EQUALP :BG COLOR.AT :XCOR :YCOR
END

TO COLOR.AT :XCOR :YCOR
OP PIXEL (.EXAMINE POSADDR :XCOR ►
    :YCOR) REMAINDER :XCOR 4
END

TO POSADDR :XCOR :YCOR
OP 16384 + 40*YCOR + INT (:XCOR/4)
END

TO PIXEL :BYTE :XCOR
IF :XCOR=0 [OP INT (:BYTE/64)]
IF :XCOR=1 [OP REMAINDER INT ►
    (:BYTE/16) 4]
IF :XCOR=2 [OP REMAINDER INT (:BYTE/4) ►
    4]
OP REMAINDER :BYTE 4
END

TO DOT :XCOR :YCOR
DOTA POSADDR :XCOR :YCOR
END

TO DOTA :ADDR
RUN SE (WORD "DOT REMAINDER :XCOR 4) ►
    .EXAMINE :ADDR
END

TO DOT0 :BYTE
.DEPOSIT :ADDR SUM (REMAINDER :BYTE ►
    64) 64*PEN
END

TO DOT1 :BYTE
.DEPOSIT :ADDR (SUM (64*INT ►
    (:BYTE/64)) (16*PEN) (REMAINDER ►
    :BYTE 16))
END

TO DOT2 :BYTE
.DEPOSIT :ADDR (SUM (16*INT ►
    (:BYTE/16)) (4*PEN) (REMAINDER ►
    :BYTE 4))
END

TO DOT3 :BYTE
.DEPOSIT :ADDR SUM (4*INT (:BYTE/4)) ►
    :PEN
END

```



```

TO FILL.RAY :XCOR :YCOR :DELTA
IF BYTEPOS :XCOR :DELTA [IF ►
  :BGBYTE=.EXAMINE POSADDR :XCOR ►
  :YCOR [OP FILL.CHUNK :XCOR :YCOR ►
  POSADDR :XCOR :YCOR :DELTA] ]
IF EDGE :XCOR :YCOR [OP :XCOR-:DELTA]
DOT :XCOR :YCOR
OP FILL.RAY :XCOR+:DELTA :YCOR :DELTA
END

TO FILL.CHUNK :XCOR :YCOR :ADDR :DELTA
.DEPOSIT :ADDR :PENBYTE
IF :BGBYTE=.EXAMINE :ADDR+:DELTA [OP ►
  FILL.CHUNK :XCOR+4*:DELTA :YCOR ►
  :ADDR+:DELTA :DELTA]
OP FILL.RAY :XCOR+4*:DELTA :YCOR ►
  :DELTA
END

TO BYTEPOS :XCOR :DELTA
IF :DELTA>0 [OP 0=REMAINDER :XCOR 4]
OP 3=REMAINDER :XCOR 4
END

TO FIND.BG :XCOR :YCOR :LIMIT
IF :XCOR>:LIMIT [OP []]
IF BYTEPOS :XCOR 1 [IF ►
  :PENBYTE=.EXAMINE POSADDR :XCOR ►
  :YCOR [OP FIND.BG :XCOR+4 :YCOR ►
  :LIMIT] ]
IF NOT EDGE :XCOR :YCOR [OP :XCOR]
OP FIND.BG :XCOR+1 :YCOR :LIMIT
END

```

Savepict and Loadpict

When you've drawn a complicated picture, it's useful to be able to save the picture itself in a disk file, so that you can later restore it to the screen without going through the procedures that drew the picture again. For example, suppose you're writing a video adventure game in which characters in the story are drawn against a backdrop showing a forest, dungeon, or whatever. The backdrop could be saved as a picture file and then loaded onto the screen for each scene before drawing in the actors.

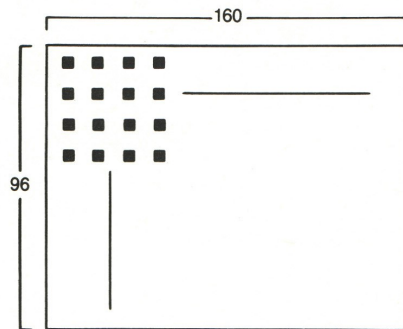
In this project, you'll see three different sets of Logo programs for saving and loading pictures. The three versions differ in how fast they can load a picture and also differ somewhat in flexibility. The last version, for example, allows a small picture to be "stamped" on the screen in different positions. One thing to learn from this project is how using different *data representations* can affect the efficiency of a program.

There are two ways to approach this project. If you just want to use these procedures as a tool to save and load pictures for some other project of your own, you don't have to understand some of the details explained here about how pictures are stored. On the other hand, by studying how the project works, you can learn about the important idea of data representation.

Note: If you have a 16K Atari computer, you should use the number 8192 instead of 16384 in procedures SAVEPICT, LOADPICT, and PICTLOC. (PICTLOC appears only in the third version of the project.) With a 16K machine, you don't have a disk drive, but you could save pictures on cassette.

How a Picture Is Stored

In order to save and load pictures, we have to know something about how a picture is represented in the Atari computer. In this project we are concerned only with the pictures drawn with pens, not with the turtle shapes. The lines you draw are represented as a pattern of dots (called *pixels*) on the screen. There are 96 rows and 160 columns of dots on the screen:



Screen pixels

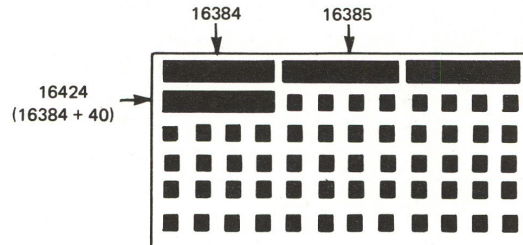
The reason that a diagonal line comes out jagged on the screen is that it isn't actually drawn as a smooth line, but simply by filling in certain dots on the screen. Each pixel can be in one of four conditions: it can be empty (that is, it can be in the background color) or it can be filled in with one of the three possible pens.

By the way, the length of a "turtle step" is not the same as the distance between pixels. That is, when you type the command `FORWARD 100` the turtle does not move 100 pixels on the screen. How many pixels it actually does move depends on the direction. If you're moving horizontally (heading 90, for example), then `FORWARD 100` moves through 50 pixels. If you're moving vertically, the distance depends on the *aspect ratio*, which is controlled by the `.SETSCR` command. The usual aspect ratio is 0.8, in which case `FORWARD 100` moves 40 pixels (50 times 0.8). In this project, since we're interested in saving a picture that is already on the screen rather than drawing a picture with turtle commands, we have to think in terms of pixels, not in terms of turtle steps.

I said that each pixel can be in any of four conditions (background or three pens). Therefore, each pixel can be represented in the computer's memory using two *bits*, or binary digits. Each bit can be either zero or one. The four conditions are represented this way:

```
0 0 background
0 1 pen 0
1 0 pen 1
1 1 pen 2
```

Memory is grouped into *bytes* of eight bits. So each byte represents four pixels. There are 96 times 160, or 15,360, pixels altogether on the screen. The memory required is one fourth of that, or 3840 bytes. It happens that the first byte of Logo's screen memory is at memory location number 16384. So the picture memory is arranged something like this:



Picture memory

Characters (letters, digits, spaces, and so on) are represented in the computer's memory by a number that is stored in one byte. For example, the letter *A* is represented by a byte containing the number 65. Most of the time you don't have to worry about this, but if you remember this fact, it'll help you understand the process of storing information in disk files.

Representing the Screen in a Disk File

The most straightforward way to represent a screen picture in a disk file is simply to write each of the 3840 bytes into the file. To find out what is in each byte, we use the `.EXAMINE` operation, which outputs a number representing the byte at whatever memory location is used as its input. For example:

```
PRINT .EXAMINE 16384
```

will print the number in the first byte of Logo's screen memory. This byte represents the first four pixels in the upper left corner of the screen. (For Atari computers with 16K of RAM, the first byte of screen memory is in location 8192 instead of 16384.)

It would be possible to save a picture in a file, then, with a program like this:

```
TO SAVEPICT :FILE
SETWRITE :FILE
SAVEPICT1 16384 3840
SETWRITE []
END

TO SAVEPICT1 :LOC :NUM
IF :NUM=0 [STOP]
PRINT .EXAMINE :LOC
SAVEPICT1 :LOC+1 :NUM-1
END
```

Each byte of the picture memory would be represented in the file by a line containing the digits in the number in that byte. That is, if a particular byte happened to contain the number 125, that byte would be stored in the file as the three digits 1, 2, 5, just as it is typed on the screen by a `PRINT` command. Each digit takes up one byte in the file. Therefore, using this scheme, it takes three bytes in the file to represent one byte in the picture!

(Actually, another byte is used to represent the end-of-line code.) This leads to very large files.

Instead, it would be better to use only one byte in the file to represent each byte in the picture. This can be done by using the operation CHAR. This procedure takes a number as its input and outputs the single character that corresponds to that number. For example, CHAR 65 outputs the letter A. Using this procedure, we can write the program as follows:

Savepict/Loadpict, Version 1

```

TO SAVEPICT :FILE
SETWRITE :FILE
SAVEPICT1 16384 3840
SETWRITE []
END

TO SAVEPICT1 :LOC :NUM
IF :NUM=0 [STOP]
TYPE CHAR .EXAMINE :LOC
SAVEPICT1 :LOC+1 :NUM-1
END

TO LOADPICT :FILE
SETREAD :FILE
LOADPICT1 16384 3840
SETREAD []
END

TO LOADPICT1 :LOC :NUM
IF :NUM=0 [STOP]
.DEPOSIT :LOC ASCII RC
LOADPICT1 :LOC+1 :NUM-1
END

```

To use the SAVEPICT procedure, you first draw a picture on the screen using the usual turtle commands. Then you say

SAVEPICT "D:PICTFILE

or whatever you want to name the file. The program writes 3840 bytes into this file. Later, you can restore the picture to the screen by typing

LOADPICT "D:PICTFILE

The operation ASCII, which is used in LOADPICT1, is the inverse of CHAR. It takes a single character as input and outputs the number that represents that character. So ASCII "A outputs 65.

Experiment with these procedures. You'll find that both saving and loading pictures are quite slow. This is because the procedures SAVEPICT1 and LOADPICT1 are invoked 3840 times, once for each byte of screen memory, even if nothing is drawn in that part of the screen. Also, the files written by this version of SAVEPICT are rather large (3840 bytes), so you can't fit very many on a diskette.

Sparse Data Representations

A typical turtle graphics picture is *sparse*. This means that most of the pixels on the screen are unused (background color), which means that most of the bytes of picture memory are zero. It seems silly to write a file that is mostly full of zeros. By using a cleverer representation of the picture, we can write smaller files and make the loading of a picture file much faster.

The idea is this: as we look through the picture memory, we'll find a bunch of zero bytes, and then a nonzero one, and then a bunch more zero bytes, and so on. To make this more specific, consider this sample fragment of a picture memory:

```
0 0 0 0 0 0 0 0 0 23 0 0 0 0 47 0 0 0 0 0 0 0 0 15
```

In the first version of the program, we'd represent these twenty-four bytes of screen memory as twenty-four bytes in the file. But instead, we can think of this as 9 zeros, 23, 4 zeros, 47, 8 zeros, 15. We could store this information in a file in this form:

```
9 23 4 47 8 15
```

In other words, we have decided that odd-numbered bytes in the file represent how many consecutive zero bytes are in the picture, while even-numbered bytes represent actual picture data. By representing the picture in this way, we've reduced twenty-four bytes of picture to six bytes in the file. We'll find that it is also much faster to load a picture stored in this form.

In practice, there may be several hundred consecutive zero bytes in a picture. This poses a slight problem: the largest number that can be represented in a single byte is 255. Therefore, if there are more than that many consecutive zeros, the new SAVEPICT procedure writes the sequence 255 0 in the file for each group of 256 zeros.

A second minor detail is that there must be a way for LOADPICT to know when the end of the file has been reached. This isn't a problem in the first version of the program because there all picture files are the same length, 3840 bytes. But in the new version, the length of the file depends on the number of pixels that are drawn in a nonbackground color. To solve this problem, SAVEPICT writes the sequence 0 0 at the end of the file. This sequence can't be part of real picture data.

Savepict/Loadpict, Version 2

```
TO SAVEPICT :FILE
  SETWRITE :FILE
  SAVEPICT1 16384 3840 0
  REPEAT 2 [TYPE CHAR 0]
  SETWRITE []
  END
```

```
TO SAVEPICT1 :LOC :NUM :NULL
  IF :NUM=0 [STOP]
  SAVEPICT1 :LOC+1 :NUM-1 SAVEPICT2 .EXAMINE :LOC :NULL
  END
```

```

TO SAVEPICT2 :BYTE :NULL
IF AND :BYTE=0 :NULL<255 [OP :NULL+1]
TYPE CHAR :NULL
TYPE CHAR :BYTE
OP 0
END

TO LOADPICT :FILE
SETREAD :FILE
LOADPICT1 16384 ASCII RC ASCII RC
SETREAD []
END

TO LOADPICT1 :LOC :NULL :BYTE
IF AND :BYTE=0 :NULL=0 [STOP]
.DEPOSIT :LOC+:NULL :BYTE
LOADPICT1 :LOC+:NULL+1 ASCII RC ASCII RC
END

```

Experiment with this version of the program. You'll notice that SAVEPICT isn't any faster, but LOADPICT is usually very much faster. The reason is that SAVEPICT must still examine every byte of picture memory, because it doesn't know ahead of time where you've drawn lines. But LOADPICT only has to deposit information into the bytes in picture memory that actually correspond to lines in the saved picture file.

Snapshots

In the second version, LOADPICT doesn't change the parts of picture memory that aren't used in the picture file you're loading. This suggests that it should be possible to *merge* two pictures. (In the first version, loading a picture file completely replaced whatever might have been on the screen before you invoked LOADPICT.) Try drawing a picture, saving it with SAVEPICT, clearing the screen, drawing another picture, and then using LOADPICT to restore the first picture. Make sure that the two pictures aren't in exactly the same part of the screen, so you can see whether the old picture remains intact.

What you'll find is that this merging of two pictures works pretty well, but not perfectly. The problem comes up if the two pictures use pixels that are right next to each other, so that a pixel in one picture is part of the same byte of memory as a pixel of the other picture. (Remember that each byte contains four pixels.) Loading a new number into that byte eliminates the pixel that used to be there. Still, this technique works perfectly if the two pictures are widely separated, and it works pretty well in most cases.

It would be handy to take advantage of this merging capability by using a picture file as a kind of rubber stamp that could be drawn in different positions on the screen. The scheme is this: you draw a small picture near the center of the screen. Then you use a version of SAVEPICT to make a "snapshot" of this picture. You can then use a version of LOADPICT to "stamp" the saved picture anywhere on the screen, depending on the turtle position.

PROGRAMMING IDEAS

To make this work, the picture file must include information about where the turtle was when the picture was taken. SAVEPICT must be modified to write this information in the file. Then LOADPICT must be modified to compare the current position of the turtle to the one stored in the file. If the two positions are different, the picture should be loaded into a different part of the screen memory.

This third version of the program is quite a bit more complicated than the others. The main reason for this is that it has to deal with the difference between pixels and turtle steps. To know where to "stamp" the saved picture in memory, we have to think in terms of pixels. But Logo tells us the turtle's position in turtle steps. This position has to be rounded off to the nearest pixel. Also, as explained earlier, the conversion between steps and pixels depends on the aspect ratio. There is no easy way for a Logo procedure to find out what this ratio is. The solution used in this program is that it looks for a variable named SCRUNCH in the workspace. If there is such a variable, its value should be the aspect ratio. If not, the standard value of 0.8 is assumed.

Another complication is that if the picture is being loaded into a position that is different from where it came from, part of the picture may extend beyond the edge of the screen. The procedure PUTBYTE in the following program is used like .DEPOSIT, but it checks to be sure that you are trying to deposit into the part of memory that contains the picture.

Savepict/Loadpict, Version 3

```

TO SAVEPICT :FILE
  IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH 0.8]
  SETWRITE :FILE
  TYPE CHAR (XCOR+160)/2
  TYPE CHAR (120-YCOR)*:SCRUNCH/2
  SAVEPICT1 16384 3840 0
  REPEAT 2 [TYPE CHAR 0]
  SETWRITE []
END

TO SAVEPICT1 :LOC :NUM :NULL
  IF :NUM=0 [STOP]
  SAVEPICT1 :LOC+1 :NUM-1 SAVEPICT2 .EXAMINE :LOC :NULL
END

TO SAVEPICT2 :BYTE :NULL
  IF AND :BYTE=0 :NULL<255 [OP :NULL+1]
  TYPE CHAR :NULL
  TYPE CHAR :BYTE
  OP 0
END

TO LOADPICT :FILE
  IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH 0.8]
  SETREAD :FILE
  LOADPICT1 PICTLOC ASCII RC ASCII RC
  SETREAD []
END

```

```

TO LOADPICT1 :LOC :NULL :BYTE
IF AND :BYTE=0 :NULL=0 [STOP]
PUTBYTE :LOC+:NULL :BYTE
LOADPICT1 :LOC+:NULL+1 ASCII RC ASCII RC
END

TO PICTLOC
OP 16384+((XDIFF ASCII RC)+160*YDIFF ASCII RC)/4
END

TO PUTBYTE :LOC :BYTE
IF (AND :LOC>16383 :LOC<20224 :BYTE>0) [.DEPOSIT :LOC :BYTE]
END

```

Note: If you have a 16K Atari computer, you should use the following:

```

IF (AND :LOC>8191 :LOC<12032 :BYTE>0) [.DEPOSIT :LOC :BYTE]

TO XDIFF :XLOC
OP INT (XCOR+160)/2-:XLOC
END

TO YDIFF :YLOC
OP INT (120-YCOR)*:SCRUNCH/2-:YLOC
END

```

To experiment with this program, try something like this:

```

CS
REPEAT 4 [FD 40 RT 90]
SAVEPICT "D:SQSNAP
PU
SETPOS [80 60]
LOADPICT "D:SQSNAP
SETPOS [-70 20]
LOADPICT "D:SQSNAP

```

In practice, you wouldn't bother making a snapshot of something as simple as a square, because it's easier to draw another square than to load it from a disk file. But if you draw more complicated pictures, in multiple colors, this technique can really be worthwhile.

Suggestion: Run-Length Encoding

What if you filled in the screen completely with some pen color and tried to save that in a picture file? Using the first version of the program, of course, it doesn't matter what's on the screen; the file ends up with 3840 data bytes. But with the two later versions, something else happens. The picture memory is completely filled with bytes that represent the same number, but not zero. For example, if you fill the screen with pen 0, the picture memory will be

```
85 85 85 85 85 ...
```


PROGRAMMING IDEAS

In the sparse encoding scheme we've been using, this is thought of this way: 0 zero bytes, 85, 0 zero bytes, 85, and so on. What ends up in the picture file is

```
0 85 0 85 0 85 0 85 0 85 ...
```

The picture file is twice as big as the screen memory! This isn't a very good result. The smart LOADPICT will be slower for this picture than the stupid one. A sparse representation only works well if the picture is, in fact, sparse.

This is an extreme, unlikely example. But it isn't unlikely for *part* of the screen to be filled in solidly. For example, if you're drawing a picture of a farm, the background might be blue to represent the sky, and there might be a large solid green area at the bottom of the screen to represent grass.

Still, although that green area isn't *empty*, it is *uniform*. The bytes representing that area in screen memory are mostly all the same, even if not all zero. We could use a slightly more complicated data representation called *run-length encoding*, which would handle this case well. Here's how it works. Instead of a two-byte sequence representing the number of zero bytes and then the value of a data byte, we can use a sequence representing the value of a data byte and the number of consecutive bytes containing that value. For example, suppose the screen memory looks like this:

```
85 85 85 85 85 1 0 0 0 0 0 0 0 43 85 85 85 85 ...
```

We would represent that in the picture file this way:

```
85 5 1 1 0 7 43 1 85 4 ...
```

In this example, the version in the file is only a little smaller than the screen memory. But in real situations, the run lengths would often be several hundred bytes, not just five or seven.

This run-length technique is often used in serious computer graphics work. It's especially efficient for black-and-white pictures, because there are only two possible values for the data. You can just alternate them and leave them out of the file. You only store the run lengths. That is, the odd-numbered bytes of the file would contain the numbers of consecutive black pixels and the even-numbered bytes would contain the numbers of consecutive white pixels.

On the other hand, for a color picture that really is sparse, the representation we've been using is somewhat more efficient than the run-length representation. The moral is that before you choose a data representation for any problem, you should think hard about different possibilities!

PROGRAM LISTING

VERSION 1

```
TO SAVEPICT :FILE
SETWRITE :FILE
SAVEPICT1 16384 3840
SETWRITE []
END
```

```
TO SAVEPICT1 :LOC :NUM
IF :NUM=0 [STOP]
TYPE CHAR .EXAMINE :LOC
SAVEPICT1 :LOC+1 :NUM-1
END
```

```

TO LOADPICT :FILE
SETREAD :FILE
LOADPICT1 16384 3840
SETREAD []
END

```

```

TO LOADPICT1 :LOC :NUM
IF :NUM=0 [STOP]
.DEPOSIT :LOC ASCII RC
LOADPICT1 :LOC+1 :NUM-1
END

```

VERSION 2

```

TO SAVEPICT :FILE
SETWRITE :FILE
SAVEPICT1 16384 3840 0
REPEAT 2 [TYPE CHAR 0]
SETWRITE []
END

```

```

TO SAVEPICT1 :LOC :NUM :NULL
IF :NUM=0 [STOP]
SAVEPICT1 :LOC+1 :NUM-1 SAVEPICT2 ►
.EXAMINE :LOC :NULL
END

```

```

TO SAVEPICT2 :BYTE :NULL
IF AND :BYTE=0 :NULL<255 [OP :NULL+1]
TYPE CHAR :NULL
TYPE CHAR :BYTE
OP 0
END

```

```

TO LOADPICT :FILE
SETREAD :FILE
LOADPICT1 16384 ASCII RC ASCII RC
SETREAD []
END

```

```

TO LOADPICT1 :LOC :NULL :BYTE
IF AND :BYTE=0 :NULL=0 [STOP]
.DEPOSIT :LOC+:NULL :BYTE
LOADPICT1 :LOC+:NULL+1 ASCII RC ASCII ►
RC
END

```

VERSION 3

```

TO SAVEPICT :FILE
IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH ►

```

```

0.8]
SETWRITE :FILE
TYPE CHAR (XCOR+160)/2
TYPE CHAR (120-YCOR)*:SCRUNCH/2
SAVEPICT1 16384 3840 0
REPEAT 2 [TYPE CHAR 0]
SETWRITE []
END

```

```

TO SAVEPICT1 :LOC :NUM :NULL
IF :NUM=0 [STOP]
SAVEPICT1 :LOC+1 :NUM-1 SAVEPICT2 ►
.EXAMINE :LOC :NULL
END

```

```

TO SAVEPICT2 :BYTE :NULL
IF AND :BYTE=0 :NULL<255 [OP :NULL+1]
TYPE CHAR :NULL
TYPE CHAR :BYTE
OP 0
END

```

```

TO LOADPICT :FILE
IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH ►
0.8]
SETREAD :FILE
LOADPICT1 PICTLOC ASCII RC ASCII RC
SETREAD []
END

```

```

TO LOADPICT1 :LOC :NULL :BYTE
IF AND :BYTE=0 :NULL=0 [STOP]
PUTBYTE :LOC+:NULL :BYTE
LOADPICT1 :LOC+:NULL+1 ASCII RC ASCII ►
RC
END

```

```

TO PICTLOC
OP 16384+((XDIFF ASCII RC)+160*YDIFF ►
ASCII RC)/4
END

```

```

TO PUTBYTE :LOC :BYTE
IF (AND :LOC>16383 :LOC<20224 :BYTE>0) ►
[.DEPOSIT :LOC :BYTE]
END

```

```

TO XDIFF :XLOC
OP INT (XCOR+160)/2-:XLOC
END

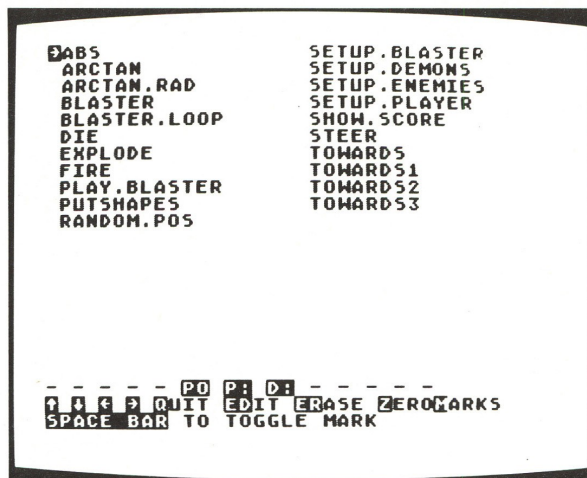
```

```

TO YDIFF :YLOC
OP INT (120-YCOR)*:SCRUNCH/2-:YLOC
END

```

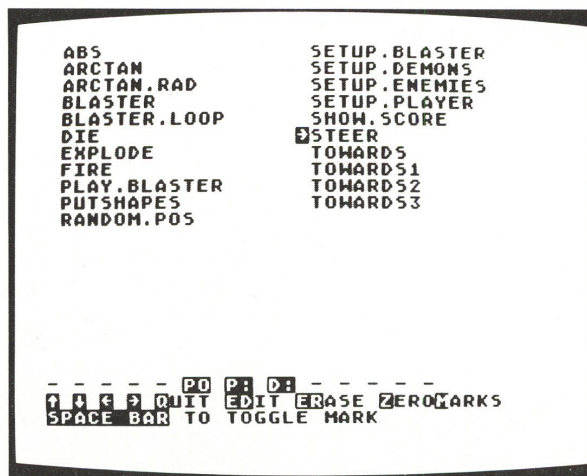

Display Workspace Manager



The Display Workspace Manager (DWM) is a tool that helps you manage projects that involve large numbers of procedures. The program lists all your procedures on the screen. You can move a pointer around, marking particular procedures. Then you can edit, erase, print, or save the marked procedures.

DWM divides the screen into two parts. The top part is used to list the names of procedures. The bottom few lines remind you of the commands you can type to DWM. (For example, you can type ER to *erase* procedures.)

In the figure above, DWM is being used to examine Blaster, a project in this book. The arrow points to the word STEER on the screen. STEER is the name of one of the procedures in Blaster. The pointer arrow can be moved from one procedure name to another by using the arrow keys on the Atari keyboard.



By Brian Harvey.

In the next figure, the user has typed the P0 command to DWM. DWM's P0 command tells it to print out the definition of the procedure at which the arrow points, in this case STEER.

```

TO STEER :WHERE
IF :WHERE < 0 [STOP]
SETH 45 * :WHERE
SETSH 1 + :WHERE
END

TYPE A SPACE WHEN READY

```

In the following figure, seven procedures have been *marked* with asterisks on the screen.

```

*ABS          SETUP.BLASTER
*ARCTAN        SETUP.DEMONS
*ARCTAN.RAD    SETUP.ENEMIES
BLASTER        SETUP.PLAYER
BLASTER.LOOP   SHOW.SCORE
DIE            STEER
EXPLODE        *TOWARDS
FIRE           *TOWARDS1
PLAY.BLASTER   *TOWARDS2
PUTSHAPES      *TOWARDS3
RANDOM.POS

```

- - - - - P0 P1 D1 - - - - -
 1 1 1 1 1 QUIT EDIT ERASE ZERO MARKS
 SPACE BAR TO TOGGLE MARK

In the next figure the user has typed the command ER, which means to erase all the marked procedures. DWM has printed "Really erase 7 proce-

PROGRAMMING IDEAS

dures?" on the bottom line of the screen. It asks this question to make it harder for someone to erase many procedures accidentally.

```

*ABS          SETUP.BLASTER
*ARCTAN       SETUP.DEMONS
*ARCTAN.RAD   SETUP.ENEMIES
BLASTER      SETUP.PLAYER
BLASTER.LOOP  SHOW.SCORE
DIE          STEER
EXPLODE      *TOWARDS
FIRE         *TOWARDS1
PLAY.BLASTER *TOWARDS2
PUTSHAPES    *TOWARDS3
RANDOM.POS

- - - - - PO P: D: - - - - -
↑ ↓ ← → QUIT EDIT ERASE ZERO MARKS
SPACE BAR TO TOGGLE MARK
REALLY ERASE ? PROCEDURES?

```

In the next figure, the user has typed Y for yes, and DWM has erased the marked procedures. It now displays a shorter list of the remaining procedures.

```

BLASTER
BLASTER.LOOP
DIE
EXPLODE
FIRE
PLAY.BLASTER
PUTSHAPES
RANDOM.POS
SETUP.BLASTER
SETUP.DEMONS
SETUP.ENEMIES
SETUP.PLAYER
SHOW.SCORE
STEER

- - - - - PO P: D: - - - - -
↑ ↓ ← → QUIT EDIT ERASE ZERO MARKS
SPACE BAR TO TOGGLE MARK

```

This is a large project. I won't attempt a complete explanation of every detail of the program. Instead, I'll indicate the most important parts to understand.

Creating the List of Procedures

In order for DWM to work, it must have a list of the names of all the procedures in your project. This list must be in a global variable named

ALL.PROCEDURES. If this list doesn't already exist, the first thing DWM does is to call DWM.PROCLIST to create the list. This automatic creation of the list requires a disk drive with a writeable disk in it! DWM.PROCLIST works by doing a POTS command while writing to the disk, then rereading the results to find the names of your procedures. When the list is created automatically, it is sorted alphabetically. The sorting process is quite slow, because it's done simply rather than cleverly. (Read the Mergesort project for another sorting technique.) The automatically generated list omits all procedures whose names start "DWM" so that the procedures in the DWM program itself won't clutter up your list.

If you want to save time when starting up DWM, or if you want the procedures in your project listed in some order other than alphabetical, you can create the variable ALL.PROCEDURES yourself and make it part of the workspace file.

How DWM Arranges the Display

Once the list of procedures exists, DWM lists them on the screen. This is done by two main procedures, DWM.SIZE.MENU and DWM.DRAW.MENU. The first of these figures out how the names should be arranged on the screen, given the number of procedures you have in your list. The more procedures, the more columns on the screen will be required to list them all. The more columns, the less wide each column can be. This limits the length of a procedure name that can be displayed. Therefore, the program uses the smallest number of columns that will fit your list. Then the DWM.DRAW.MENU procedure uses this information to draw the display.

If the name of a procedure is too long to fit in a screen column, an inverse video plus sign (+) is shown at the end of the truncated name.

Reading DWM Commands

The procedure DWM.MAIN.LOOP reads and processes the commands you type to the program. Commands are either one or two characters long. Here is a list of the commands.

arrows	Move the pointer up, down, left, or right. You can type the arrow keys either with or without the CTRL key held down.
space bar	<i>Mark</i> the procedure where the pointer is, if it's not marked already, or unmark it if it is. An asterisk is displayed next to the name of marked procedures.
ED	Edit the marked procedures in the Logo editor.
ER	Erase all the marked procedures. This command first tells you how many procedures are marked and insists that you type Y to confirm that you really want to erase the procedures.
P0	Print out on the screen the single procedure whose name is pointed to by the arrow.
P:	List all marked procedures on the printer.
D:	Save all marked procedures on the disk. This command prompts for a filename to be used for the saved procedures. Notice that these save files do not contain the values of varia-

PROGRAMMING IDEAS

bles! But the procedures in them can be loaded with the LOAD command.

ZM Zero Marks. Unmark all procedures.
Q Quit. Exits from DWM.

If there are no marked procedures, the commands that normally apply to marked procedures apply instead to all procedures in the display. Be careful about erasing!

Possible Extensions

DWM takes up just under 2000 nodes, somewhat more than half the available space. This limits the size of the programs you can use it on. (This is particularly unfortunate since it's the big projects that most need this sort of help.)

If there were space, this project could be the basis for implementing workspace management tools like PACKAGE and BURY, which are found in some other versions of Logo. The technique would be to have several lists of procedures instead of just one ALL.PROCEDURES list.

PROGRAM LISTING

In the program listing that follows, characters that are underlined represent inverse-video characters on the Atari.

```
TO DWM
DWM.1 [] [] [] [] [] []
END

TO DWM.1 :PROCS :COLUMNS :CHARS :ROWS :TABS :MARKED
IF NOT NAMEP "ALL.PROCEDURES [DWM.PROCLIST]
DWM.SIZE.MENU
DWM.DRAW.MENU
SETCURSOR [0 0]
DWM.SHOW.CURSOR 1
DWM.MAIN.LOOP RC 0 1 1
END
```

CREATING THE ALL.PROCEDURES LIST

```
TO DWM.PROCLIST
PR [ONE MOMENT, I'M LISTING...]
SETWRITE "D:DWM.TMP
POTS
SETWRITE []
SETREAD "D:DWM.TMP
PR [HANG ON A BIT LONGER...]
MAKE "ALL.PROCEDURES []
DWM.READ.TITLE RL
SETREAD []
ERF "D:DWM.TMP
END
```

```
TO DWM.READ.TITLE :LINE
IF EMPTY :LINE [STOP]
DWM.READ.TITLE1 FIRST BF :LINE
DWM.READ.TITLE RL
END
```

```
TO DWM.READ.TITLE1 :NAME
IF EQUALP "DWM DWM.FIRSTPART :NAME 3 [STOP]
MAKE "ALL.PROCEDURES DWM.INSERT :NAME :ALL.PROCEDURES
END
```

```
TO DWM.INSERT :WORD :LIST
IF EMPTY :LIST [OP FPUT :WORD []]
IF DWM.BEFORE :WORD FIRST :LIST [OP FPUT :WORD :LIST]
OP FPUT FIRST :LIST DWM.INSERT :WORD BF :LIST
END
```

```
TO DWM.BEFORE :NEW :OLD
IF EMPTY :NEW [OP "TRUE]
IF EMPTY :OLD [OP "FALSE]
IF (ASCII FIRST :NEW) < (ASCII FIRST :OLD) [OP "TRUE]
IF (ASCII FIRST :NEW) > (ASCII FIRST :OLD) [OP "FALSE]
OP DWM.BEFORE BF :NEW BF :OLD
END
```

```
TO DWM.FIRSTPART :WORD :NUM
IF EMPTY :WORD [OP "]
IF EQUALP :NUM 1 [OP FIRST :WORD]
OP WORD FIRST :WORD DWM.FIRSTPART BF :WORD :NUM-1
END
```

PRINTING THE MENU

```
TO DWM.SIZE.MENU
MAKE "PROCS COUNT :ALL.PROCEDURES
MAKE "COLUMNS 1+INT ((:PROCS-1)/20)
MAKE "CHARS (INT 37/:COLUMNS)-2
MAKE "ROWS 1+INT ((:PROCS-1)/:COLUMNS)
MAKE "TABS DWM.SIZE.TABS 1 :CHARS+2 :COLUMNS
END
```

```
TO DWM.SIZE.TABS :COL :CHARS :COLS
IF :COLS = 0 [OP [99]]
OP FPUT :COL DWM.SIZE.TABS :COL+:CHARS :CHARS :COLS-1
END
```

```
TO DWM.DRAW.MENU
TS CT
DWM.DRAW.M1 :ALL.PROCEDURES 0 1 BF :TABS 1
SETCURSOR [0 20]
PR [- - - - - PO P: D: - - - - -]
(PR CHAR 156 CHAR 157 CHAR 158 CHAR 159
 [QUIT EDIT ERASE ZEROMARKS])
PR [SPACE BAR TO TOGGLE MARK]
END
```


PROGRAMMING IDEAS

```

TO DWM.DRAW.M1 :PROCS :ROW :COL :TABS :INDEX
IF EMPTY :PROCS [STOP]
IF MEMBERP :INDEX :MARKED [DWM.STAR]
SETCURSOR LIST :COL :ROW
TYPE DWM.SHORT FIRST :PROCS :CHARS
IF EQUALP :ROW :ROWS-1 ►
  [DWM.DRAW.M1 BF :PROCS ►
    Ø FIRST :TABS BF :TABS :INDEX+1] ►
  [DWM.DRAW.M1 BF :PROCS :ROW+1 :COL :TABS :INDEX+1]
END

```

```

TO DWM.STAR
SETCURSOR LIST :COL-1 :ROW
TYPE "*"
END

```

```

TO DWM.SHORT :NAME :CHARS
IF (COUNT :NAME)<(:CHARS+1) [OP :NAME]
OP DWM.SHORT1 :NAME :CHARS
END

```

```

TO DWM.SHORT1 :NAME :CHARS
IF :CHARS=1 [OP CHAR 171]
OP WORD FIRST :NAME DWM.SHORT BF :NAME :CHARS-1
END

```

READING COMMANDS FROM THE KEYBOARD

```

TO DWM.MAIN.LOOP :CMD :ROW :COL :INDEX
DWM.SET.CURSOR
IF :CMD = "-" [DWM.UP]
IF :CMD = CHAR 28 [DWM.UP]
IF :CMD = "=" [DWM.DOWN]
IF :CMD = CHAR 29 [DWM.DOWN]
IF :CMD = "+" [DWM.LEFT]
IF :CMD = CHAR 30 [DWM.LEFT]
IF :CMD = "*" [DWM.RIGHT]
IF :CMD = CHAR 31 [DWM.RIGHT]
IF :CMD = CHAR 32 [DWM.TOGGLE.MARK]
IF :CMD = "E" [DWM.CMD.2 "E [[D DWM.EDIT] [R DWM.ERASE]]]
IF :CMD = "Z" [DWM.CMD.2 "Z [[M DWM.FLUSH.MARKS]]]
IF :CMD = "P" ►
  [DWM.CMD.2 "P [[O DWM.PRINTOUT] [: DWM.PRINTER]]]
IF :CMD = "D" [DWM.CMD.2 "D [[: DWM.DISKSAVE []]]]
IF :CMD = "Q" [SETCURSOR [Ø 23] STOP]
DWM.HIDE.CURSOR
SETCURSOR LIST (DWM.ITEM :COL :TABS)-1 :ROW
MAKE "INDEX (:COL-1)*:ROWS+:ROW+1
DWM.SHOW.CURSOR :INDEX
DWM.MAIN.LOOP RC :ROW :COL :INDEX
END

```

```

TO DWM.ITEM :NUM :LIST
IF :NUM=1 [OP FIRST :LIST]
OP DWM.ITEM :NUM-1 BF :LIST
END

```

```
TO DWM.CMD.2 :LETTER :LIST
DWM.PROMPT :LETTER
DWM.CMD.21 RC :LIST
DWM.SET.CURSOR
END
```

```
TO DWM.CMD.21 :CHAR :LIST
DWM.PROMPT CHAR 32
IF EMPTYP :LIST [TOOT 0 400 10 10 STOP]
IF EQUALP :CHAR FIRST FIRST :LIST [RUN BF FIRST :LIST STOP]
DWM.CMD.21 :CHAR BF :LIST
END
```

```
TO DWM.PROMPT :LETTER
SETCURSOR [0 23]
TYPE :LETTER
END
```

```
TO DWM.SHOW.CURSOR :INDEX
TYPE CHAR IF MEMBERP :INDEX :MARKED [170] [159]
END
```

```
TO DWM.HIDE.CURSOR
TYPE CHAR IF MEMBERP :INDEX :MARKED [42] [32]
END
```

```
TO DWM.SET.CURSOR
SETCURSOR LIST (DWM.ITEM :COL :TABS)-1 :ROW
END
```

MOVING THE POINTER

```
TO DWM.RIGHT
IF (:COL*:ROWS+:ROW+1) > :PROCS [TOOT 0 400 10 10 STOP]
MAKE "COL :COL+1
END
```

```
TO DWM.LEFT
IF :COL=1 [TOOT 0 400 10 10 STOP]
MAKE "COL :COL-1
END
```

```
TO DWM.DOWN
IF :INDEX+1 > :PROCS [TOOT 0 400 10 10 STOP]
IF :ROWS > :ROW+1 [MAKE "ROW :ROW+1 STOP]
MAKE "ROW 0
MAKE "COL :COL+1
END
```

```
TO DWM.UP
IF :ROW>0 [MAKE "ROW :ROW-1 STOP]
IF :COL=1 [TOOT 0 400 10 10 STOP]
MAKE "ROW :ROWS-1
MAKE "COL :COL-1
END
```


PROGRAMMING IDEAS**SETTING AND CLEARING MARKS**

```

TO DWM.TOGGLE.MARK
IF MEMBERP :INDEX :MARKED [DWM.UNMARK] [DWM.MARK]
END

```

```

TO DWM.MARK
MAKE "MARKED FPUT :INDEX :MARKED
END

```

```

TO DWM.UNMARK
MAKE "MARKED DWM.REMOVE :INDEX :MARKED
END

```

```

TO DWM.REMOVE :THING :LIST
IF EMPTY? :LIST [OP []]
IF EQUALP :THING FIRST :LIST [OP BF :LIST]
OP FPUT FIRST :LIST DWM.REMOVE :THING BF :LIST
END

```

```

TO DWM.FLUSH.MARKS
MAKE "MARKED []
DWM.DRAW.MENU
END

```

EDIT

```

TO DWM.EDIT
EDIT DWM.MARKLIST
DWM.DRAW.MENU
END

```

```

TO DWM.MARKLIST
IF EMPTY? :MARKED [OP :ALL.PROCEDURES]
OP DWM.MARKLIST1 :ALL.PROCEDURES 1
END

```

```

TO DWM.MARKLIST1 :LIST :NUM
IF EMPTY? :LIST [OP []]
IF MEMBERP :NUM :MARKED [OP FPUT FIRST :LIST
    DWM.MARKLIST1 BF :LIST :NUM+1]
OP DWM.MARKLIST1 BF :LIST :NUM+1
END

```

PRINTOUT

```

TO DWM.PRINTOUT
CT
PO DWM.ITEM :INDEX :ALL.PROCEDURES
TYPE [TYPE A SPACE WHEN READY]
DWM.IGNORE RC
DWM.DRAW.MENU
END

```

```

TO DWM.IGNORE :CHAR
END

```

ERASE

```

TO DWM.ERASE
DWM.PROMPT (SE [REALLY ERASE] ►
COUNT DWM.MARKLIST [PROCEDURES?])
IF NOT EQUALP RC "Y [DWM.CLEAR.PROMPT STOP]
DWM.CLEAR.PROMPT
ERASE DWM.MARKLIST
DWM.ERASE1 DWM.MARKLIST
MAKE "MARKED []
DWM.SIZE.MENU
DWM.DRAW.MENU
MAKE "ROW 0
MAKE "COL 1
END

TO DWM.ERASE1 :LIST
IF EMPTY :LIST [STOP]
MAKE "ALL.PROCEDURES DWM.REMOVE FIRST :LIST :ALL.PROCEDURES
DWM.ERASE1 BF :LIST
END

TO DWM.CLEAR.PROMPT
DWM.PROMPT CHAR 32
REPEAT 35 [TYPE CHAR 32]
END

```

SAVE TO D: OR P:

```

TO DWM.DISKSAVE :FILE
DWM.PROMPT "FILE:
MAKE "FILE FIRST RL
CT
SETWRITE WORD "D: :FILE
DWM.SAVE DWM.MARKLIST
SETWRITE []
DWM.DRAW.MENU
END

TO DWM.PRINTER
CT
SETWRITE "P:
DWM.SAVE DWM.MARKLIST
SETWRITE []
DWM.DRAW.MENU
END

TO DWM.SAVE :LIST
IF EMPTY :LIST [STOP]
PO FIRST :LIST
DWM.SAVE BF :LIST
END

```